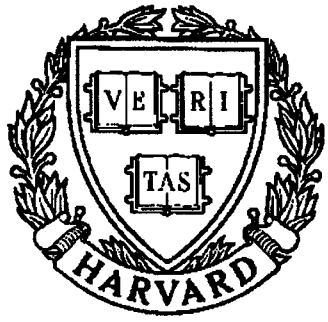
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), the University of Maryland, Harvard University, and Industry

Client/Server Model for Distributed Computing: An Implementation

by A. Sela

TR 89-7

Client/Server model for distributed computing: An Implementation

Amir Sela
Electrical Engineering Department
&
Systems Research Center
University of Maryland

1 Introduction

At the Intelligent Servosystems Laboratory work has been proceeding for some time on the development of a distributed processing environment to allow for the implementation of complex dynamics simulations.[†] Initially, these simulations were written on a Silicon Graphics *IRIS 3130* workstation with integrated numerical and graphics display code. While this approach is satisfactory for computationally non-intensive applications, it became evident that more computing power would become necessary in order to achieve simulation in real time. The logical step was to use more powerful computers for the number crunching, and to retain the *IRIS* workstation, which is optimized for various graphics tasks, such as rotating, filling, clipping, etc. for display purposes. This necessitated the development of code to allow for interprocess communication among various machines. Our computing environment consists of Sun and *IRIS* workstations, connected via an Ethernet.

[†]This work was supported in part by the National Science Foundation's Engineering Research Centers Program: NFSD CDR 8803012, by the NSF-REU program of the Systems Research Center, by AFOSR-URI grant AFOSR-87-0073, and by a NASA-USRA Design project grant.

2 The Client/Server model

The simulation is first divided into two stages: the numerical and the graphics display. The numerical code is compiled on the SUN, while the graphics code is delegated to the *IRIS*. A 'client/server' arrangement is used, in which two communication sockets are opened between the two cooperating machines. The *server* runs on the SUN, listening for connection requests on a known, predetermined port. When the user wishes to initiate a simulation, the *client* is invoked on the *IRIS*. All user interaction is taken care of by the *IRIS*, complete with pop-up menus and dialogue boxes. When run, the client will rendezvous with the server at the predetermined port, and a pair of sockets will be established. Although it is possible to use one socket for two-way communication, it was decided to utilize each socket as a one way communication device. This is convenient in that it facilitates the creation of a 'token ring' arrangement, where a data structure is passed around a ring between machines until it reaches its destination. As the structure makes its way from machine to machine, each computer examines the destination host field of the structure, compares it to its own, and decides to accept the data if it is addressed to it, or pass it along to the next machine.

2.1 The Server

Three different approaches to the server/numerical process where implemented. They differ substantially in terms of system architecture, but require little change in computer code. The first implementation requires one process, which is responsible for the numerical simulations as well as for monitoring the communication link for client requests. We refer to this as the *polling server* approach. This is perhaps the most intuitive approach, although the least efficient. The second approach, although it requires two processes, frees the numerical process from having to check if there is data from the client at each iteration. This approach implements an interrupt mechanism to inform the numerical process that data is pending. Finally, the third and most efficient method is a server which uses interrupt driven socket I/O to inform the process that data is present. This has the advantage of not requiring two processes, and of using lower level constructs for efficiency. We will refer to two-process server as the *pseudo-interrupt driven* server, while the third method will be referred to as the *interrupt driven* server.

2.1.1 Polling Server

In this approach, the server opens a communication socket for writing when it is first invoked. It then listens for connection requests. When a request is detected, it opens a socket for reading, and is ready to begin the simulation. The program is organized so that before beginning each iteration, the *read* socket is

polled to see if data has been sent by the client. If data has indeed been sent, then it is read and acted upon. Otherwise, the simulation continues with the next iteration.

2.1.2 Pseudo Interrupt Driven Server

In the second approach, program flow is identical to the polling server for the connection establishment phase. Once the connection is established, the process issues a *fork()* call, spawning a new version of itself. This new version (the child) becomes the numerical simulation, while the old (the parent) listens at the *read* socket for incoming data. If data is detected, the parent sends the child an interrupt, informing the numerical process that the user wishes it to perform some action. These actions may be simple start or stop commands, or they may set the simulation's initial conditions. It is important to note that while the parent process is waiting for commands from the client, the numerical process continues with its calculations.

2.1.3 Interrupt Driven Server

This server model is the culmination of the experience gained with the two models previously discussed. This approach is the fastest and most efficient. It requires only one process, unlike the *pseudo-interrupt* driven server, and does not need to poll the socket to see if data is present, like the polling server. This makes is much faster and more elegant than either of the other alternatives. The use of the Unix *signal* interrupt feature allows us to utilize interrupt driven socket I/O. [1].

2.2 The Client

When the client is invoked, it attempts to rendezvous with the server running on a foreign host. If the server is found to be executing, then the client may establish the communication link between the two machines. Once the connection has been established, the user is left with a *read* socket and a *write* socket. That is, whenever commands are to be sent to the simulation server, they are simply written to the client's 'write' socket, and whenever the client wishes to read the data generated by the server, it simply polls its 'read' socket to see if data is available. It is up to the user to determine the commands that are to be recognized by the client/server pair, and to implement the computer code that handles them.

2.3 Data Representation

The simplest, and perhaps the most inefficient method of sending data between machines is to convert to 'char' (ASCII) representation, send the string of characters across the network, and then convert back to machine representation.

When programming in C this involves doing an *sprintf* into a string, writing that string across the network, and using *sscanf* to recover the data at the other end. Not only is one forced to send a larger number of characters, across the network, but one must also spend the time parsing, formatting, and so forth. The overhead associated with this method makes it prohibitive for all but the simplest applications. Rather than pursue this approach, we chose to exploit the fact that both the *IRIS*es and Suns have the same word length, and both store data in an identical manner, with one exception, which we will discuss. Instead of writing an ASCII string, we write the data structure itself across the network. This relieves us of the time consuming task of building a string with all of the data on the sending end, and of parsing that data on the receiving end. This yields a dramatic increase in data throughput. Clearly, one may *not* include pointers in the structure that is to be written, as this would simply transfer the address of the data on the *other* machine, and not the data itself.

2.4 External Data Representation

It is important to remember that not all machines store data in the same format. The data transfer method we employ works reliably, if one is careful to make sure that the data representation of the machines is compatible. This is the case for the Suns and *IRIS*es, with one minor exception, namely, whenever one wishes to send a variable which is type 'double' on the Sun, it must be called a 'long float' on the *IRIS*. In order to overcome these data compatibility problems, a set of library routines called *xdr* [3] may be used. This *External Data Representation* allows one to transfer data across machines without regard as to how they represent data. At this point in time we do not make use of these procedures, although they will likely be utilized in the future.

2.5 Data Transfer

One very important aspect of implementing the simulation is the design of the necessary data structures which must be passed between machines. One may opt to have one large structure which contains all of the data necessary to construct a frame at a given timestep, or one may wish to send several smaller structures for each frame. Both methods have particular advantages and drawbacks. Sending one large structure is convenient as it simplifies the implementation of the simulation in some high-level language. When one reads data, one has *a priori* knowledge of the kind of structure which was sent. Its major disadvantage is that nonessential data is often sent in the data structure along with data which is essential for the display of the scene. Sending a variety of smaller data structures is more efficient in terms of the amount of data which is sent across the network, but involves considerably more programming effort. The method of data transfer presented here has been seen to provide a peak transfer rate of 20 kbytes/second. This was arrived at by removing the numerical

code from the simulations and writing a non-changing data structure across the network.

2.6 Animation Considerations

Real-time animation demands the generation of natural appearing sequences of images. To the human observer, a rate of thirty frames per second will meet this goal; however, even rates as low as twenty frames per second will present an acceptable sense of continuity to the observer. Taking thirty frames per second as a specification for the animation allows 33.3 msec of processing time to be apportioned to the tasks of performing calculations and displaying the next frame. On the *IRIS* workstation, one buffer clear requires 12 msec. This leaves us 21.3 msec in which to perform our calculations and draw the scene. For complicated simulations on the *IRIS*, it is only through the use of distributed processing that this can be achieved.

3 Inverted Pendulum Simulation

The inverted pendulum on a cart is the classic 'broom balancing' control problem. The idea is for the user to sit at the *IRIS* graphics workstation, enter some feedback gains which define the control law, and watch in living color as the cart moves back and forth, trying to stabilize the pendulum. Although the simulation is not so numerically intensive that it could not be implemented entirely on the *IRIS*, it was decided to use this as a simple case study for testing the client/server model with a real simulation.

3.1 Client: Sending commands to the numerical process

Generally, the user will wish to set certain initial conditions, gains, and so forth before starting the simulation. In our simple example, we only need to be able to send floating point data, while in more complicated examples, we require flexible data structures to handle varying data types and structures. A simple procedure call is used to send commands to the remote processor. This is done as follows:

send_sim_command(command,value);

where command is an integer constant, consistent on both machines, and value is the new value to be passed to the simulation process.

The procedure send_sim_command looks like this:

```
send_sim_command(type,val)
int type;
float val;
{
  struct send_cmd
       {
       int type;
       float val;
  };
  struct send_cmd data_rec;

  data_rec.type=type;   /* Load 'type' into data structure */
  data_rec.val=val;     /* Load 'val' into data structure  */

  if ((write(wsock,&data_rec,sizeof(data_rec))) < 0 {
       perror("Writing on stream socket");
       exit(1);
  }
}
```

3.2 Client: Receiving data from the numerical process

In order to receive data from the numerical process, one must first poll the 'read' socket to see if data is available to be read. Attempting to read data from a socket on which there is no data pending must be avoided, as it may cause the program to hang. Therefore, one must first issue a *select* call to determine if data is present. The following code fragment is recommended for reading data from a socket.

```
read_next_data()
{
fd_set read_template;
struct timeval wait;

  wait.tvsec = 0;
  wait.tvusec = 0;
  FD_ZERO(&read_template);
  FD_SET(rsock,&read_template);
  select(FD_SETSIZE, &read_template, (fd_set *) 0, (fd_set *) 0, &wait);
  if (FD_ISSET(rsock,&read_template)) {
      if (read(rsock,&data_back,sizeof(data_back)) <= 0 ) {
        perror("reading from stream socket");
        exit(1);
      }
  }
}
```

Here 'data_back' is the structure in which the results of the calculations are returned.

3.3 Server: Receiving Commands from the Client

As described earlier, three different versions of the server were written. These are the *polling* server, the *pseudo-interrupt* driven server, and the real *interrupt driven* server. We now describe the various server implementations.

3.3.1 Polling Server

In this approach, the server polls it's 'read' socket before each iteration of the numerical algorithm. This is accomplished by using a variation of the 'read_next_data' routine discussed above. It may be preferable to place the code inline to avoid the extra procedure call, as it is in a critical loop. If data is present, it is read, acted upon, and the simulation continues. Some sample code follows:

```
while (CONTINUE) {
  /* Check for data from client */
  FD_ZERO(&read_template);
  FD_SET(rsock,&read_template);
  select(FD_SETSIZE, &read_template, (fd_set *) 0, (fd_set *) 0, &wait);
  if (FD_ISSET(rsock,&read_template)) {
       if (read(rsock,&data_back,sizeof(data_back)) <= 0 ) {
         perror("reading from stream socket");
         exit(1);
       }

  /* Perform some action based on the contents of 'data_back' */
  }

 /* Perform next iteration of numerical algorithm */
}
```

3.3.2 Pseudo-Interrupt driven server

This approach offers a significant improvement over the polling server. As discussed earlier, once the client and server rendezvous, the server spawns a processing using *fork()*. The parent process monitors the 'read' socket for data from the client, while the numerical process is free to compute the next iteration and to send them to the client. When the server parent detects that data has become available from the client, it sends an interrupt to the child. On receipt of the interrupt, the numerical process will execute an interrupt handler, which will set a flag 'INTERRUPTED' to TRUE. Program execution will resume so that the calculation of the current iteration will be completed. Before beginning the next iteration, the numerical program will check if there is data pending (by examining the flag 'INTERRUPTED') and will read and process it if this is the case.

```
Set_Signal_handler
while (DONT_STOP) {
     while (INTERRUPTED || !INITIALIZED) {
       Check for incoming data
       If data present, read and act in it.
       set INTERRUPTED to FALSE
     }
     while (!INTERRUPTED && INITIALIZED) {
       Compute next iteration
```

```
        }
    }
```

The following lines of code are necessary to set the signal handler to trap interrupts. Whenever data is sent from the client to the server, the numerical process (the child) will receive an interrupt from its parent.

```
int onintr(),(*istat)();
istat=signal(SIGINT,SIG_IGN); /* save original status */
if (istat!=SIG_IGN) {
      signal(SIGINT,onintr);
}

onintr()
{
  signal(SIGINT,onintr);  /* Reset for next interrupt */
  INTERRUPTED=TRUE;
}
```

3.3.3 Interrupt driven server

This approach is only slightly different from the *pseudo-interrupt* driven server, although it is much more elegant. Here we require only the numerical process, and do not need to concern outselves with checking for incoming data. The receipt of incoming data is made known to the child process via a SIGIO signal. The server must set up an interrupt handler and trap this signal. The code that does this follows:

```
signal(SIGIO, onintr);

/* Set the process receiving SIGIO/SIGURG signals to us */

if (fcntl(rmsgsock, F_SETOWN, getpid()) < 0) {
        perror("fcntl F_SETOWN");
        exit(1);
}

/* Allow receipt of asynchronous I/O signals */

if (fcntl(rmsgsock, F_SETFL, FASYNC) < 0) {
        perror("fcntl F_SETFL, FASYNC");
        exit(1);
```

}

3.3.4 Why the Pseudo-Interrupt Driven Server Architecture?

After developing the architecture and implementation of the *interrupt driven* server, one could ask "Why bother with this implementation? The real interrupt driven server is so much more elegant." Although for applications which involve a graphics client and a single numerical server this is indeed the case, there remain applications where the *pseudo-interrupt driven* server is more appropriate. One may wish, for instance, to have a number of processes running on the same machine. The 'client' in this case would send commands to the various processes running on the foreign host. Rather than interrupt *each* process when data becomes is available, the pseudo interrupt server would read the data, and selectively send an interrupt to the correct process. Thus for process management, a supervisory pseudo-interrupt driven server may be most appropriate.

3.4 Server: Sending Data to the Client

When the simulation process wishes to send data to the client, the following code fragment is used. The structure 'send_rec' contains the data to be sent. Further, more robust error handling provisions will be developed in the future.

```
if ((write(wsock,&send_rec, sizeof(send_rec))) < 0) {
     perror("writing on stream socket");
     exit(0);
}
```

In our pendulum example, the send_rec data structure consists of:

```
struct data_rec
{
float time;
float k1;
float k2;
float x1;
float x2;
float z1;
float z2;
float deltat;   /* timestep */
float M1;       /* mass of bob */
float M2;       /* mass of cart */
float L;        /* Length of pendulum */
float force;
};
```

Note that the data structure consists of all of the data required to characterize the state of the system at any one time. For the case of the inverted pendulum, we choose to pass all of the data at one time to simplify program structure. At the end of each iteration, the results are stored in send_rec and written to the client. This enables the graphics display program to alternately change the quantities to be plotted (at the request of the user) without having to request that the numerical process (server) send different data structures. Although it would be slightly faster to have only the necessary data sent across the network, it has been found that this is of no consequence for this simulation, as the 'bottleneck' appears to be in the area of graphics display, where data generated by the Sun workstation arrives at the *IRIS* graphics workstation more quickly than the *IRIS* can display the results.

4 Server Performance Comparisons

We compare the data transfer rate of the three server architectures discussed. This is done using the inverted pendulum simulation as a test case. In order to benchmark the server, and not the client graphics program running on the *IRIS* workstation, we chose to eliminate the graphics code and instead see how much data becomes available at the *IRIS* end. By eliminating the graphics, we insure a more reliable evaluation of the server architecture. The benchmark program has the following structure:

```
For (i=0;i<NUM_TRIALS;i++) {
        record start time
        send 'START'
        for (j=0;j<NUM_ITERATIONS;j++) { check_for_data }
        send 'PAUSE'
        record end time
        flush ipc buffer
        display time interval, compute kbytes/sec, etc.
}
```

Running the pendulum simulation with the same initial conditions under the same system load, we arrive at the following results:

Polling Server

time (sec)	iterations	Bytes	Kbytes/sec	flushed
11.733	179	8592	0.732	3
11.767	177	8496	0.722	0
11.783	182	8736	0.741	0
11.783	180	8640	0.733	0
11.783	180	8640	0.733	0
11.717	180	8640	0.737	1
11.700	176	8448	0.722	0
11.767	184	8832	0.751	0
11.750	177	8496	0.723	3

Pseudo-Interrupt Driven Server

time (sec)	iterations	Bytes	Kbytes/sec	flushed
14.217	3232	155136	10.912	13
14.450	3267	156816	10.852	14
15.333	3606	173088	11.288	0
14.217	3153	151344	10.646	9
14.233	3224	154752	10.873	11
14.050	3005	144240	10.266	6
13.950	2914	139872	10.027	1
14.150	3113	149424	10.560	11
13.950	2900	139200	9.978	11

Interrupt Driven Server

time (sec)	iterations	Bytes	Kbytes/sec	flushed
15.250	4396	211008	13.837	11
15.300	4459	214032	13.989	11
15.150	4277	205296	13.551	2
16.600	4847	232656	14.015	12
17.367	5055	242640	13.972	0
16.750	4678	224544	13.406	0
17.450	4830	231840	13.286	0
17.133	5014	240672	14.047	4
17.033	4897	235056	13.800	11

5 Future Work

There remains much work to be done in the development of the client/server model for distributed processing. The generalization of this approach to create a parallel processing environment, in which many engineering workstations work in parallel is our ultimate goal. This paper has presented both an overview of the client/server model, as well as a detailed example of its implementation in a simple distributed simulation. In the future, more robust error-handling routines must be written, and more ambitious simulations, incorporating many workstations, should be attempted.

Acknowledgments

I would like to thank Dr. P.S. Krishnaprasad for his inspiration and many discussions on the subject of distributed processing. I would also like to thank Russ Byrne, with whom I worked closely. Many thanks are due to all of the faculty and students of the Intelligent Servosystems Laboratory, for their continuing support and encouragment.

References

[1] Joy, W.; Fabry, R.; Leffler, S. *A 4.3BSD Interprocess Communication Primer*, Computer Systems Research Group, University of California, Berkeley. Berkeley, CA, 94720.

[2] Sun Microsystems. *SUN IPC Tutorial*, Mountain View, CA, 1987.

[3] Sun Microsystems Inc. *SUN External Data Representation*, Mountain View, CA, 1987.

Server Architectures